-7826

ANL-7826

CHINESE REMAINDER AND INTERPOLATION ALGORITHMS

John D. Lipson

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Prepared for the U.S. ATOMIC ENERGY COMMISSION
under contract W-31-109-Eng-38

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy $3.00; Microfiche $0.95

ANL-7826
Mathematics and
Computers

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

CHINESE REMAINDER AND INTERPOLATION ALGORITHMS*

by

John D. Lipson

Applied Mathematics Division

July 1972

*This research was supported by the U. S. Atomic Energy Commission and the National Research Council of Canada.

TABLE OF CONTENTS

LIST OF TABLES

CHINESE REMAINDER AND INTERPOLATION ALGORITHMS

by

John D. Lipson

ABSTRACT

This paper is concerned with mathematical, computational, and historical aspects of the Chinese Remainder and Interpolation Theorems of number theory and numerical analysis, with a view to their application to symbolic computation.

1. INTRODUCTION

The great usefulness of modular arithmetic and interpolation methods as tools for exact and symbolic computation is becoming increasingly recognized (e.g. cf. [3, 5, 14, 18]). The Chinese Remainder and Interpolation Theorems from number theory and numerical analysis play a key role in these methods; the Chinese Remainder Theorem allows for the reconstruction of an integer from its residues with respect to an appropriate number of moduli, and interpolation formulae allow for the reconstruction of a polynomial from an appropriate number of sample values.

This paper attempts a study in depth of the mathematical and algorithmic aspects of the Chinese Remainder and Interpolation Theorems, with a view to their application to problems involving computation with (large) integers and/or polynomials with integral coefficients.

In Sec. 2, a Chinese Remainder Theorem is established in the suitably abstract setting of an arbitrary commutative ring. The special feature of the proof of this theorem is the identification of two Chinese Remainder Formulas, called "Lagrangian" and "Newtonian," having distinct computational properties.

In Sec. 3, various algorithms based on the Chinese Remainder Formulas of Sec. 2 are derived in the more specialized algebraic setting of abstract Euclidean domains. A generalization (to Euclidean domains) of the

divided-differences associated with Newton's (polynomial) Interpolation Formula is also given.

In Sec. 4, the algebraic setting is further specialized to the domains of primary importance for symbolic computation: the integers $\underline{Z}$ and polynomials F[x] with coefficients in a field F. Examples are given to illustrate and contrast the various algorithms in both of these cases.

Sec. 5 is devoted to the analysis and appraisal of the various Chinese Remainder Algorithms, in order to answer the important question "which one should be used in practice?"

In Sec. 6, an attempt is made to summarize the interesting and somewhat complicated history of the Chinese Remainder and Interpolation algorithms studied in this paper, updating previous accounts by the inclusion of material relevant to symbolic computation.

2. AN ABSTRACT CHINESE REMAINDER THEOREM

The algebraic setting for this section of the paper is as follows. Let K be a commutative ring with unit element 1, and $M = \{M_0, M_1, \ldots, M_n\}$ a finite set of ideals in K. A special property of the set of ideals M that we shall need is given by the following

Definition. The set $M = \{M_0, M_1, \ldots, M_n\}$ of distinct ideals in a commutative ring K is called a *pair-wise spanning* (PS) system of ideals if

$$M_i + M_k = k \qquad (\forall i \neq k). \tag{2.1}$$

As an immediate consequence of this definition we have the following

Lemma 2.1. If M is a PS system of ideals in K, then for any pair of ideals M_i, $M_k \in M$ $(i \neq k)$ there exist $n_i^{(k)} \in M_i$, $n_k^{(i)} \in M_k$ such that

$$n_i^{(k)} + n_k^{(i)} = 1 \quad . \tag{2.2}$$

Anticipating future development, it shall be convenient to assume a factorization for the element $n_i^{(k)}$ of the form

$$n_i^{(k)} = s_i^{(k)} m_i^{(k)} \quad , \tag{2.3}$$

where $s_i^{(k)} \in K$, $m_i^{(k)} \in M_i$. Note that there is no loss of generality in assuming such a factorization, for if $n_i^{(k)} \in M_i$ we can always take $s_i^{(k)} = 1$ and $m_i^{(k)} = n_i^{(k)}$. Conversely, if $s_i^{(k)} \in K$ and $m_i^{(k)} \in M_i$ then $s_i^{(k)} m_i^{(k)} \in M_i$ by the defining properties for an ideal.

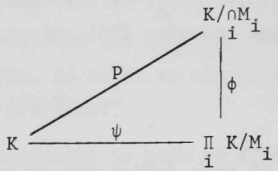
Theorem 2.2 (Abstract Chinese Remainder Theorem). Let K be a commutative ring, and $M = \{M_0, M_1, \ldots, M_n\}$ a PS system of ideals in K. Then the map

$$\phi: u + \bigcap_{i=0}^{n} M_i \to (u+M_0, \ldots, u+M_n) \tag{2.4}$$

is an isomorphism of rings

$$K/\cap_i M_i \simeq \prod_i K/M_i \quad .^*$$

Proof. Consider the following mapping diagram



with ϕ as in (2.4), p the natural map from K to its quotient-ring $K/\cap_i M_i$, and $\psi: K \to \prod_i K/M_i$ defined by $u \to (u+M_0,\ldots,u+M_n)$. It is easily verified that ψ is a morphism with kernel $\cap_i M_i$. It follows by the Isomorphism Theorem for Rings that ϕ is an isomorphism from $K/\cap_i M_i$ to $\psi(K)$.

Still to be proved is that ψ is surjective (and hence that ϕ is an isomorphism from $K/\cap_i M_i$ to $\prod_i K/M_i$). Thus, for arbitrary $(u_0+M_0,\ldots, u_n+M_n) \in \prod_i K/M_i$ we must establish $u \in K$ such that $\psi(u) = (u_0+M_0,\ldots, u_n+M_n)$. An equivalent but more intuitive statement of the same problem is given by the following.

The Chinese Remainder Problem. For arbitrary $u_i \in K$, find a solution to the system of congruences

$$u \equiv u_i \pmod{M_i} \qquad (i=0,1,\ldots,n). \qquad (2.5)$$

In the context of the ring of integers $\underline{Z}$, finding a solution to the system of congruences (2.5) (where M_i is some principal ideal (m_i) and each u_i may be regarded as a remainder or residue when the (unknown) integer u is divided by m_i) is the celebrated Chinese Remainder Problem

---

*Unless specified otherwise, indices are assumed to range over $\{0,1,\ldots,n\}$. Thus $\cap_i$ stands for $\cap_{i=0}^{n}$, $\prod_i$ for $\prod_{i=0}^{n}$.

[16, pp. 57-64] of Number Theory (thus the name of Theorem 2.2 and the above problem).

The main result of this section now follows, in the establishment of two "Chinese Remainder" formulas for determining a solution u to the system of congruences (2.5) under the conditions of Theorem 2.2: (i) a Lagrangian formula, and (ii) a Newtonian formula. The reason for this nomenclature will be evident when we consider these formulas in special cases.

(i) Lagrangian Formula

With $s_i^{(k)}$, $m_i^{(k)}$ as in (2.3), let

$$L_k = \Big| \prod_{i\neq k} s_i^{(k)} \Big|_{M_k} \times \prod_{i\neq k} m_i^{(k)} \qquad (k=0,1,\ldots,n). \tag{2.6}$$

Then the Lagrangian formula for u is

$$u = \sum_{k=0}^{n} u_k L_k \quad . \tag{2.7}$$

The validity of (2.7) is established by

Theorem 2.3. $u \equiv u_k \pmod{M_k}$.

Proof. From (2.6), each L_k is of the form $\alpha \prod_{i\neq k} m_i^{(k)}$, with $\alpha \in K$, $m_i^{(k)} \in M_i$. Hence, by the defining properties of an ideal, each $L_k \in \bigcap_{i\neq k} M_i$, whence

$$L_k \equiv 0 \pmod{M_i} \qquad (\forall i\neq k).$$

Also,

$$\begin{aligned} L_i &= \Big| \prod_{i\neq k} s_i^{(k)} \Big|_{M_k} \times \prod_{i\neq k} m_i^{(k)} \\ &\equiv \prod_{i\neq k} s_i^{(k)} m_i^{(k)} \pmod{M_k} \\ &\equiv \prod_{i\neq k} (1-n_i^{(k)}) \pmod{M_k} \quad . \end{aligned}$$

Now from (2.2) each $n_i^{(k)} \in M_k$, so that

$$L_k \equiv 1 \pmod{M_k} \quad .$$

Thus we have shown that

$$L_k \equiv \delta_{ik} \pmod{M_i}$$

and the statement of the theorem follows immediately.

(ii) Newtonian Formula

Again with $s_i^{(k)}$, $m_i^{(k)}$ as in (2.3), a sequence $u^{[0]},u^{[1]},\ldots,u^{[n]} \in K$ is defined recursively by

$$u^{[k]} = u^{[k-1]} + a_k \prod_{i=0}^{k-1} m_i^{(k)} \quad , \tag{2.8}$$

where

$$a_0 = u^{[0]} = |u_0|_{M_0} \quad , \tag{2.9}$$

$$a_k = |(u_k - u^{[k-1]}) \times \prod_{i=0}^{k-1} s_i^{(k)}|_{M_k} \quad . \tag{2.9'}$$

Then the Newtonian formula for u is

$$u = u^{[n]} = a_0 + a_1 m_0^{(1)} + \ldots + a_n \prod_{i=0}^{n-1} m_i^{(n)} \quad . \tag{2.10}$$

As for the Lagrangian formula, we must prove

Theorem 2.4. $u \equiv u_k \pmod{M_k}$.

Proof. By induction we show that

$$u^{[k]} \equiv u_i \pmod{M_i} \qquad (i=0,1,\ldots,k).$$

For k = 0 we have $u^{[0]} \equiv u_0 \pmod{M_0}$ by (2.9). Assuming

$$u^{[k-1]} \equiv u_i \pmod{M_i} \qquad (i=0,1,\ldots,k-1),$$

it follows that

$$u^{[k-1]} + \alpha_k\beta_k \equiv u_i \pmod{M_i} \qquad (i=0,1,\ldots,k-1)$$

for any $\alpha_k \in K$, $\beta_k \in \bigcap_{i=0}^{k-1} m_{k-1}$. The particular choice $\alpha_k = a_k$, $\beta_k = \prod_{i=0}^{k-1} m_i^{(k)}$ gives

$$u^{[k]} = u^{[k-1]} + a_k \prod_{i=0}^{k-1} m_i^{(k)}$$

$$= u^{[k-1]} + \left|(u_k - u^{[k-1]}) \times \prod_{i=0}^{k-1} s_i^{(k)}\right|_{M_k} \times \prod_{i=0}^{k-1} m_i^{(k)}$$

$$\equiv u^{[k-1]} + (u_k - u^{[k-1]}) \times \prod_{i=0}^{k-1} s_i^{(k)} m_i^{(k)} \pmod{M_k} \quad .$$

Now $\prod_{i=0}^{k-1} s_i^{(k)} m_i^{(k)} = \prod_{i=0}^{k-1} (1-n_k^{(i)}) \equiv 1 \pmod{M_k}$ because from (2.2) $n_k^{(i)} \in M_k$, whence from the above

$$u^{[k]} \equiv u^{[k-1]} + (u_k - u^{[k-1]}) \times 1 \pmod{M_k}$$

$$\equiv u_k \pmod{M_k} \quad .$$

Thus $u^{[k]} \equiv u_i \pmod{M_i}$ $(i=0,1,\ldots,k)$, which completes the proof by induction. For $k = n$ we obtain

$$u = u^{[n]} \equiv u_i \pmod{M_i} \qquad (i=0,1,\ldots,n),$$

which completes the proof of the theorem.

Referring to the proof of Theorem 2.2, either the Lagrangian or Newtonian Chinese Remainder Formula establishes that $\psi\colon K \to \prod_i K/M_i$ is surjective and hence that ϕ of (2.4) is an isomorphism of rings $K/\cap_i M_i \simeq \prod_i K/M_i$. This completes the proof of Theorem 2.2.

As for *uniqueness* properties of a solution u to the system of congruences (2.5), we have by the Isomorphism Theorem for Rings that

$$\psi(u) = \psi(u') \Rightarrow u \equiv u' \pmod{\operatorname{Ker}\psi} \quad .$$

But, as already noted, ψ has kernel $\cap_i M_i$, so that a solution to (2.5) is unique modulo the ideal $\cap_i M_i$.

Thus, both the Lagrangian and Newtonian Chinese Remainder Formulas must yield essentially the same solution to a system of congruences. However, algorithms based on these formulas have quite different computational properties with respect to time and storage requirements. Our goal now is the derivation and analysis of such algorithms in algebraic settings relevant to symbolic computation.

3. CHINESE REMAINDER ALGORITHMS IN ABSTRACT EUCLIDEAN DOMAINS

In this section we derive algorithms based on the Lagrangian and Newtonian Chinese Remainder Formulas (2.7) and (2.10) for solving the Chinese Remainder Problem in a Euclidean domain D. Specifically, this is the problem of computing the solution u to a system of congruences

$$u \equiv u_i \pmod{m_i} \qquad (i=0,1,\ldots,n), \tag{3.1}$$

with arbitrary $u_i \in D$ and moduli m_i pairwise relatively prime, i.e., $(m_i,m_k) = 1 \ \forall i \neq k$.

The Chinese Remainder Theorem and Formulas of Sec. 2 apply to the above system of congruences by virtue of

Lemma 3.1. The principal ideals $(m_0),(m_1),\ldots,(m_n)$ constitute a PS system of ideals in D.

This is an immediate consequence of

Euclid's Lemma. If $(m_i,m_k) = 1$ then the equation

$$xm_i + ym_k = 1 \tag{3.2}$$

has a solution $x = s_i^{(k)}$, $y = s_k^{(i)}$ in D (cf. (2.2) and (2.3)). Furthermore, the solution $s_i^{(k)}$, $s_k^{(i)}$ is computable by the "extended" Euclidean algorithm [11, p. 302]. For future reference we note from (3.2) that

$$s_i^{(k)} m_i \equiv 1 \pmod{m_k} \tag{3.3}$$

(and, symmetrically, $s_k^{(i)} m_k \equiv 1 \pmod{m_i}$).

According to the uniqueness considerations discussed at the end of Sec. 2, a solution to the system of congruences (3.1) is unique only up to the ideal $\bigcap_i (m_i)$. Now by assumption the moduli m_i are pairwise relatively prime, so that $\bigcap_i (m_i)$ is the (principal) ideal $(\prod_i m_i)$. Thus a solution u to (3.1) is unique modulo the product of the moduli m_i.

With $s_i^{(k)}$ as above, (2.6 - 2.7) become

Lagrangian Chinese Remainder Formula in Euclidean Domains.

$$L_k = \Big| \prod_{i \neq k} s_i^{(k)} \Big|_{m_k} \times \prod_{i \neq k} m_i \tag{3.4}$$

with

$$u = \sum_k u_k L_k \quad . \tag{3.5}$$

Thus, the specialization to Euclidean domains admits a considerable simplification in the general Lagrangian Formula (2.6 - 2.7) in that the $m_i^{(k)}$'s of that formula are now independent of k, with each $m_i^{(k)} = m_i$.

Turning to the Newtonian case, (2.8 - 2.10) become

Newtonian Chinese Remainder Formula in Euclidean Domains.

$$u^{[k]} = u^{[k-1]} + a_k \prod_{i=0}^{k-1} m_i \quad , \tag{3.6}$$

where

$$a_0 = u^{[0]} = |u_0|_{m_0} \quad , \tag{3.7}$$

$$a_k = \Big| (u_k - u^{[k-1]}) \times \prod_{i=0}^{k-1} s_i^{(k)} \Big|_{m_k} \quad , \tag{3.7'}$$

with the Newtonian Formula proper for u being

$$u = u^{[n]} = a_0 + a_1 m_0 + \ldots + a_n \prod_{i=0}^{n-1} m_i \quad . \tag{3.8}$$

As in the Lagrangian case, the simplification afforded by the specialization to Euclidean domains is that the $m_i^{(k)}$'s of (2.8 - 2.10) are now independent of k.

It is convenient to regard the system of congruences (3.1) as specifying the *modular representation* of the solution u (with respect to the moduli $m_0, m_1, \ldots, m_n$), which we denote by

$$u = [u_0, u_1, \ldots, u_n] \quad . \tag{3.9}$$

The modular representation $[u_0,u_1,\ldots,u_n]$ determines u uniquely modulo $\prod m_i$, as already noted.

Similarly, the a_i's of (3.3) can be regarded as specifying the *Newtonian representation* of u (again with respect to $m_0,m_1,\ldots,m_n$), and we denote (3.3) by

$$u = \langle a_0,a_1,\ldots,a_n \rangle \quad . \tag{3.10}$$

Thus, the problem of finding a solution to the system of congruences (3.1) by the Newtonian Formula can be regarded as one of *conversion of representations*--from the modular representation (3.9) to the Newtonian representation (3.10).

We now investigate the computational details of the Newtonian Formula. From (3.6) and (3.7), we have

$$a_0 = |u_0|_{m_0} \quad , \tag{3.11}$$

$$a_1 = |[u_1-a_0] \times s_0^{(1)}|_{m_1} \quad , \tag{3.11'}$$

$$\vdots$$

$$a_k = \left|\left[u_k-(a_0+a_1m_0+\ldots+a_{k-1}\prod_{i=0}^{k-2} m_i)\right] \times \prod_{i=0}^{k-1} s_i^{(k)}\right|_{m_k} \quad . \tag{3.11''}$$

Applying Horner's polynomial evaluation to the term (...) in (3.11") gives

$$a_k = \left[u_k-((\ldots((a_{k-1}m_{k-2}+a_{k-2})m_{k-3}+a_{k-3})m_{k-4}+\ldots)m_0+a_0)\right] \times \prod_{i=0}^{k-1} s_i^{(k)}\Big|_{m_k} \quad . \tag{3.12}$$

Thus the conversion $u = [u_0,u_1,\ldots,u_n] \rightarrow u = \langle a_0,a_1,\ldots,a_n \rangle$ can be accomplished by the following algorithm, expressed as an Algol-like program. The constants

$$c_k = \left|\prod_{i=0}^{k-1} s_i^{(k)}\right|_{m_k} \qquad (k=1,2,\ldots,n) \tag{3.13}$$

are assumed available (precomputed) for use in the program.

Algorithm N1. (Conversion from modular to Newtonian representation in a Euclidean domain)

```
begin
    a_0 := |u_0|_{m_0} ;
    for k := 1 step 1 until n do
    begin
        t := a_{k-1} ;
        for i := k-2 step -1 until 0 do
            t := |t × m_i + a_i|_{m_k} ;
        a_k := |(u_k-t) × c_k|_{m_k}
    end
end;
```

Another computational variant of the Newtonian Formula is obtained by distributing the product $\prod_{i=0}^{k-1} s_i^{(k)}$ through the bracketed [ ] expression of (3.11"), applying (3.3) and finally Horner's polynomial evaluation scheme to give

$$a_k = |(\ldots(((u_k-a_0)s_0^{(k)}-a_1)s_1^{(k)}-a_2)s_2^{(k)}-\ldots-a_{k-1})s_{k-1}^{(k)}|_{m_k}. \quad (3.14)$$

This leads to a second algorithm for carrying out the conversion $u = [u_0,u_1,\ldots,u_n] \rightarrow u = \langle a_0,a_1,\ldots,a_n \rangle$.

Algorithm N2.

```
begin
   a_0 := |u_0|_{m_0} ;
   for k := 1 step 1 until n do
   begin
      t := |u_k|_{m_k} ;
      for i := 0 step 1 until k-1 do
          t := |(t-a_i) × s_i^(k)|_{m_k} ;
      a_k := t
   end
end;
```

Multiplied Differences in Euclidean domains. We now embellish the description of the above algorithm in introducing the notion of "multiplied differences." (As shall become evident, these multiplied differences generalize to arbitrary Euclidean domains the divided differences used in the construction of Newton's Interpolating Polynomial.)

Relative to the system of congruences (3.1), the *multiplied differences* of orders 0,1,...,i ($i \leq n$) are defined by

$$[m_k] = |u|_{m_k} \qquad (0 \leq k \leq n), \tag{3.15}$$

$$[m_0, m_k] = |([m_k] - [m_0]) \times s_0^{(k)}|_{m_k} \qquad (1 \leq k \leq n), \tag{3.15'}$$

and, recursively,

$$[m_0, m_1, \ldots, m_{i-1}, m_k]^* = |([m_0, m_1, \ldots, m_{i-2}, m_k] - [m_0, m_1, \ldots, m_{i-1}]) \times s_{i-1}^{(k)}|_{m_k} \qquad (i \leq k \leq n). \tag{3.15''}$$

---

*Although the notation for multiplied differences is the same as that for the modular representation (3.9), it will always be clear from context which is intended.

These multiplied differences are arranged in a *table of multiplied differences* as follows, illustrated for n=3.

Table 3.1. Table of Multiplied Differences

[ ]	[,]	[, ,]	[, , ,]
$[m_0]$			
$[m_1]$	$[m_0,m_1]$		
$[m_2]$	$[m_0,m_2]$	$[m_0,m_1,m_2]$	
$[m_3]$	$[m_0,m_3]$	$[m_0,m_1,m_3]$	$[m_0,m_1,m_2,m_3]$

The relationship between multiplied differences and the values computed by Algorithm N2 is given by

Theorem 3.2. $[m_0,m_1,\ldots,m_{i-1},m_k]$

$$= |(\ldots(((u_k-a_0)s_0^{(k)}-a_1)s_1^{(k)}-a_2)s_2^{(k)}-\ldots-a_{i-1})s_{i-1}^{(k)}|_{m_k} .$$

Setting i = k and comparing with (3.14) yields the immediate

Corollary. $[m_0,m_1,\ldots,m_{k-1},m_k] = a_k$.

Proof. The proof of the theorem follows readily by induction on the order of the multiplied differences. For multiplied differences of order zero the theorem becomes $[m_k] = |u_k|_{m_k}$, which holds by (3.15). Now assume the theorem holds for multiplied differences of order i-1. Then by (3.15") we have

$$[m_0,m_1,\ldots,m_{k-1},m_k]$$
$$= |([m_0,m_1,\ldots,m_{i-2},m_k] - [m_0,m_1,\ldots,m_{i-1}]) \times s_i^{(k)}|_{m_k} .$$

Observing that the multiplied differences appearing on the r.h.s. are of order i-1, we apply the induction hypothesis to the first divided difference and the induction hypothesis along with (3.14) to the second divided difference, obtaining

$$[m_0,m_1,\ldots,m_{k-1},m_k]$$
$$= |(\{((\ldots((u_k-a_0)s_0^{(k)}-a_1)s_1^{(k)}-\ldots-a_{i-2})s_{i-2}^{(k)})\} - \{a_{i-1}\})s_{i-1}^{(k)}|_{m_k},$$

which completes the proof by induction of the theorem.

Thus the above theorem identifies the k-th row of the table of multiplied differences with the partial results in the computation of a_k according to (3.14). Specifically, the auxiliary variable t in Algorithm N2 successively takes on the values (for fixed k)

$$[m_k],\ [m_0,m_k],\ [m_0,m_1,m_k],\ldots,[m_0,m_1,\ldots,m_{k-1},m_k].$$

Thus Algorithm N2 computes the multiplied differences of Table 3.1 row by row but retains only the diagonal entries--these are the desired a_k's in the Newtonian Chinese Remainder Formula.

Yet a third computational variant of the Newtonian Formula is obtained directly from (3.6 - 3.7') and is carried out according to Algorithm N3 below. As in Algorithm N1, the constants c_k of (3.13) are assumed available. In addition, it is assumed that the moduli products

$$q_k = \prod_{i=0}^{k-1} m_i \qquad (k=1,2,\ldots,n) \tag{3.16}$$

are also precomputed.

Algorithm N3

```
begin
   U := |u_0|_{m_0} ;
   for k = 1 step 1 until n do
   begin
      v := |U|_{m_k} ;
      a := |(u_k-v) × c_k|_{m_k} ;
      U := U + a × q_k
   end
end;
```

Thus Algorithm N3 computes the $u^{[k]}$'s of (3.6), with the variable U successively taking on the values $u^{[0]}, u^{[1]}, \ldots,$ and finally $u^{[n]}$, which is the desired solution according to (3.8).

On the other hand, Algorithms N1 and N2 do not compute these $U^{[k]}$'s but instead compute only the coefficients $a_0, a_1, \ldots, a_n$ of the Newtonian representation (3.10) of the solution u. In order to obtain u explicitly (i.e., as an element in D instead of as a vector of coefficients a_i in D), Algorithms N1 and N2 must be followed by an *evaluation* of the expression (3.8), which on applying Horner's scheme becomes

$$u = (\ldots((a_n m_{n-1} + a_{n-1})m_{n-2} + a_{n-2})\ldots + a_1)m_0 + a_0. \qquad (3.17)$$

4. CHINESE REMAINDER ALGORITHMS IN $\underline{Z}$ AND F[x]

In this section we restrict our attention to the two Euclidean domains of principal computational interest: the domain F[x] of polynomials over a field F with the Euclidean degree being defined as the (polynomial) degree, and the domain $\underline{Z}$ of integers with the Euclidean degree being the absolute value.

In $\underline{Z}$, the integer Division Algorithm yields for any $a,m \in \underline{Z}$ unique integers $q,r \in \underline{Z}$ such that

$$a = qm + r \qquad (0 \leq r < m)$$

and we subsequently define

$$|a|_m = r \quad . \tag{4.1}$$

Analogously for F[x], the polynomial Division Algorithm yields for any $a(x),m(x) \in F[x]$ unique polynomials $q(x),r(x) \in F[x]$ such that

$$a(x) = q(x)m(x) + r(x) \qquad (\deg r(x) < \deg m(x))$$

and we subsequently define

$$|a(x)|_{m(x)} = r(x) \quad . \tag{4.2}$$

Thus, for apparent reasons related to efficiency of computation we have defined the modulo operators $|\ |_m$ and $|\ |_{m(x)}$ to yield a result that is "small" in terms of Euclidean degree; $|a|_m$ is the least non-negative integer in the ideal $a + (m)$, $|a(x)|_{m(x)}$ is the polynomial of least degree in the ideal $a(x) + (m(x))$.

Chinese Remainder Problem in $\underline{Z}$ and F[x]. In $\underline{Z}$ the Chinese Remainder Problem becomes: Find a solution u to the system of congruences

$$u \equiv u_i \pmod{m_i} \qquad (i=0,1,\ldots,n), \tag{4.3}$$

where $u_0,u_1,\ldots,u_n \in \underline{Z}$ are arbitrary and the moduli $m_0,m_1,\ldots,m_n \in \underline{Z}$ are pairwise relatively prime. A solution u is unique modulo $\prod_i m_i$, so it is convenient to impose the condition $0 \leq u < \prod_i m_i$ so that the solution to the system of congruences (4.3) is unique in $\underline{Z}$. As in (3.9), it is also convenient to consider (4.3) as specifying the *modular representation*

$$u = [u_0,u_1,\ldots,u_n] \tag{4.4}$$

of the (unknown) integer u (with respect to the moduli $m_0,m_1,\ldots,m_n$).

In F[x], the *Interpolation Problem* is as follows: construct a polynomial of degree $\leq n$

$$u(x) = s_0 + s_1x + \ldots + s_nx^n \tag{4.5}$$

which satisfies

$$u(x_i) = u_i \qquad (i=0,1,\ldots,n) \tag{4.6}$$

for arbitrary $u_0,u_1,\ldots,u_n \in F$ and distinct $x_0,x_1,\ldots,x_n \in F$.

This polynomial interpolation problem can be restated in the following congruential terms: Given the moduli polynomials $x-x_i$ $(i=0,1,\ldots,n)$ (which are clearly relatively prime if $x_i \neq x_j$ for $i \neq j$), find a solution u(x) to the system of congruences

$$u(x) \equiv u_i \pmod{x-x_i} \qquad (i=0,1,\ldots,n). \tag{4.7}$$

Thus we see that in F[x] the Interpolation Problem is nothing more than a *special* case of the Chinese Remainder Problem in F[x]--special in that the relatively prime moduli polynomials are all linear. (We shall consider an example of the more general case at the end of this section.)

As in the integer case, it is convenient to regard (4.6) (or (4.7)) as specifying the modular representation

$$u(x) = [u_0,u_1,\ldots,u_n] \tag{4.8}$$

of the (unknown) polynomial u(x) of (4.5).

Note that the well-known uniqueness property of the interpolating polynomial (4.5) can be established entirely within the ring-theoretic context at hand, for according to Sec. 3 (following (3.3)), a solution to (4.7) is unique modulo $\prod_{i=0}^{n} (x-x_i)$, i.e., unique modulo a polynomial of degree n+1. But no two distinct polynomials of degree $\leq n$ can be congruent modulo a polynomial of degree n+1, which establishes uniqueness.

One remark on terminology: With respect to a coefficient domain D, we refer to "the Interpolation (or Chinese Remainder) Problem in D[x]" whenever the solution polynomial (4.5) is known (from *a priori* considerations) to lie in D[x] and the evaluation points x_i are chosen in D (whence the sample values u_i of (4.6) are also in D). Of course, nothing new is involved in theory, since we may always embed D in its field of quotients Q(D) in order to obtain the Euclidean domain Q(D)[x]. However, because computation in Q(D) is typically far more complicated and time-consuming than computation in D, we shall be especially concerned with "D-closure" characteristics of our various Chinese Remainder algorithms, i.e., concerned with whether or not these algorithms involve intermediate computation in Q(D)[x] even though the solution polynomial is in D[x].

The situation wherein the solution polynomial u(x) is known to lie in D[x] arises naturally in symbolic computation, where D is typically the integers Z (or, recursively, multi-variate polynomials with coefficients in Z). For example, consider the problem of inverting a matrix A = A(x) with elements $a_{ij}(x) \in Z[x]$. Formally the solution is given by the matrix of rational functions

$$A(x)^{-1} = B(x)/d(x) \tag{4.9}$$

where $B(x) = (b_{ij}(x))$ is the adjoint of A and d(x) is the determinant of A. Now instead of computing with polynomials or rational functions one can instead compute $B(x_i)$, $d(x_i)$ for a sufficient number of "sample values" $x_i \in Z$ in order to construct the polynomials $b_{ij}(x)$, d(x) by interpolation, and hence obtain the desired inverse according to (4.9). Note in particular that the polynomials $b_{ij}(x)$, d(x) are known *a priori* to lie in Z[x]

because they all have determinantal definitions. Moreover, the number of sample values x_i that must be used in order to recover the solution can be readily determined (or bounded) in terms of the input data $a_{ij}(x)$, but we shall not consider this problem here.

Lagrangian Solution to the Chinese Remainder Problem in $\underline{Z}$ and $F[x]$.*

In $\underline{Z}$ the Lagrangian Formula for solving (3.1) takes the form (3.4 - 3.5) with the $s_i^{(k)}$'s of (3.4) computed by the extended Euclidean algorithm. Thus in the integer case there is no simplification in the Lagrangian Formula over the general Euclidean domain case. However, one point is noteworthy, namely the equivalence between the ancient Chinese Formula (e.g. cf. [16, p. 246]) for solving the system of congruences (3.1) in $\underline{Z}$ and the Lagrangian Formula (3.4 - 3.5). This ancient Chinese Formula takes the form

$$u = \sum_k u_k b_k M_k \quad , \tag{4.10}$$

where

$$M_k = \prod_{i \neq k} m_i, \quad b_k M_k \equiv 1 \pmod{m_k}. \tag{4.11}$$

Thus, in order to establish the equivalence of (4.10 - 4.11) with (3.4 - 3.5) we need only verify that $\left| \prod_{i \neq k} s_i^{(k)} \right|_{m_k}$ is a solution to the congruence $b_k M_k \equiv 1 \pmod{m_k}$. But

$$\left| \prod_{i \neq k} s_i^{(k)} \right|_{m_k} M_k \equiv \prod_{i \neq k} s_i^{(k)} m_i \pmod{m_k}$$

$$\equiv 1 \pmod{m_k} \text{ by (3.3)},$$

which establishes the equivalence.

Example 4.1. u has the modular representation u = [6, 9, 2, 13] with respect to the moduli $\underline{m}$ = (7, 11, 13, 15); i.e., $u \equiv 6 \pmod 7$, $u \equiv 9 \pmod{11}$, etc. Compute u (as a decimal integer).

---

* By the Chinese Remainder Problem in $F[x]$ we shall mean the special case (4.7) corresponding to the Interpolation Problem in $F[x]$ unless indicated otherwise.

Lagrangian solution: From (3.4) we have

$$L_0 = |(2)(6)(1)|_7 \times 2145 = 10725$$

$$L_1 = |(8)(6)(3)|_{11} \times 1365 = 1365$$

$$L_2 = |(2)(6)(7)|_{13} \times 1155 = 6930$$

$$L_3 = |(13)(11)(7)|_{15} \times 1011 = 11011$$

whence

$$u = 6(10725) + 9(1365) + 2(6930) + 13(11011)$$
$$= 233638$$

If we abide by the convention of taking $0 \leq u < \Pi_i m_i$ as constituting *the* solution the Chinese Remainder Problem, then

$$u = |233638|_{15015}$$
$$= 8413$$

For the case of the Chinese Remainder (Interpolation) Problem in F[x] (where $m_i(x) = x - x_i$ $(i=0,1,\ldots,n)$--see (4.7)), the $s_i^{(k)}$'s of (3.3) can be determined explicitly from

$$\frac{1}{x_k - x_i}(x - x_i) + \frac{1}{x_i - x_k}(x - x_k) = 1 \quad , \tag{4.12}$$

whence

$$s_i^{(k)} = \frac{1}{x_k - x_i} \quad . \tag{4.13}$$

The Lagrangian Formula (3.4 - 3.5) then becomes

$$L_k(x) = \left| \frac{1}{\prod_{i \neq k} (x_k - x_i)} \right|_{(x - x_k)} \times \prod_{i \neq k} (x - x_i)$$

$$= \frac{\prod_{i \neq k} (x - x_i)}{\prod_{i \neq k} (x_k - x_i)} \tag{4.14}$$

and

$$u(x) = \sum_k u_k L_k(x) \quad . \tag{4.15}$$

Thus our Lagrangian Chinese Remainder Formulas (2.6 - 2.7) and (3.4 - 3.5) have specialized in (4.14 - 4.15) to Lagrange's Interpolation Polynomial (thus the term "Lagrangian" that we have employed in the more general cases). Also we conclude that the ancient Chinese solution (4.10 - 4.11) to the integer Chinese Remainder Problem and Lagrange's Interpolation Polynomial (4.14 - 4.15) are *abstractly* equivalent.

Example 4.2. u(x) has the modular representation [10, 334, 1040, 5920] with respect to the moduli $\underline{m}(x) = (x-1, x-4, x-6, x-11)$; i.e., $u(1) = 10$, $u(4) = 334$, etc. Compute u(x) in the standard form (4.5).

Lagrangian solution (Lagrange Interpolation). From (4.14 - 4.15) we have

$$u(x) = 10\,\frac{x^3 - 21x^2 + 134x - 264}{-150}$$

$$+ 334\,\frac{x^3 - 18x^2 + 83x - 66}{42}$$

$$+ 1040\,\frac{x^3 - 16x^2 + 59x - 44}{-50}$$

$$+ 5920\,\frac{x^3 - 11x^2 + 34x - 24}{350}$$

$$= 4x^3 + 5x^2 - x + 2 \quad .$$

Newtonian Solution of the Chinese Remainder Problem in $\underline{Z}$ and F[x].

In $\underline{Z}$ the Newtonian Formula for solving (3.1) takes the form (3.8), which can be implemented according to Algorithm N1 (essentially (3.12)), Algorithm N2 (essentially (3.14)), or Algorithm N3 (essentially 3.6 - 3.7')). We illustrate the three methods below.

Example 4.3. Compute the solution u to the Chinese Remainder Problem of Example 4.1 (u has the modular representation [6, 9, 2, 13] with respect

to the moduli $\underline{m} = (7, 11, 13, 15)$), this time employing Newtonian methods.

The (precomputed) constants $s_i^{(k)}$, c_k, q_k used by Algorithms N1, N2, and N3 are as follows:

$$s_0^{(1)} = 8, \quad s_0^{(2)} = 2, \quad s_0^{(3)} = 13$$

$$s_1^{(2)} = 6, \quad s_1^{(3)} = 11$$

$$s_2^{(3)} = 7$$

$$c_1 = 8, \quad c_2 = 12, \quad c_3 = 11$$

$$q_1 = 7, \quad q_2 = 77, \quad q_3 = 1001$$

Method 1: Following Algorithm N1, we compute

$$a_0 = |6|_7 = 6$$

$$a_1 = |(9-6)8|_{11} = 2$$

$$a_2 = |(2 - (2\times 7+6))12|_{13} = 5$$

$$a_3 = |(13 - ((5\times 11+2)7+6))11|_{15} = 8,$$

whence by (3.10) u has the Newtonian representation

$$u = \langle 6, 2, 5, 8\rangle,$$

and finally by (3.8) and (3.17)

$$\begin{aligned} u &= 6 + 2(7) + 5(7)(11) + 8(7)(11)(13) \\ &= ((8\times 13+5)11 + 2)7 + 6 \\ &= 8413, \text{ as in Example 4.1.} \end{aligned}$$

Method 2: Following Algorithm N2, we compute

$$a_0 = |6|_7 = 6$$

$$a_1 = |(9-6)8|_{11} = 2$$

$$a_2 = |((2-6)2 - 2)6|_{13} = 5$$

$$a_3 = |(((13-6)13 - 2)11 - 5)7|_{15} = 8,$$

giving u = 8413, as above. The associated table of multiplied-differences (which essentially records the partial results in the computation of the a_k's, as discussed following the proof of Theorem 3.2) is displayed below.

Table 4.1. Multiplied-Differences of [6, 9, 2, 13] with Respect to $\underline{m}$ = (7, 11, 13, 15)

[ ]	[,]	[, ,]	[, , ,]
$6 = a_0$			
9	$2 = a_1$		
2	5	$5 = a_3$	
13	1	4	$8 = a_4$

Method 3: Following Algorithm N3, the variables v, a, U are successively computed as

k=0: U = 6

k=1: $v = |6|_{11} = 6$

$a = |(9-6)8|_{11} = 2$

U = 6 + 2 × 7 = 20

k=2: $v = |20|_{13} = 7$

$a = |(2-7)12|_{13} = 5$

U = 20 + 5 × 77 = 405

k=3: $v = |405|_{15} = 0$

$a = |(13-9)11|_{15} = 8$

U = 405 + 8 × 1001 = 8413

The final value of U is then the solution to the Chinese Remainder Problem. Note that the variable a in Algorithm N3 takes on successively the values $a_1, a_2, \ldots, a_n$ of the Newtonian representation of u, but these values are not retained.

We turn now to the Chinese Remainder (Interpolation) problem in F[x]. With moduli $m_i(x) = x - x_i$, the Newtonian Formula (3.8) becomes

$$u(x) = a_0 + a_1(x-x_0) + \ldots + a_n \prod_{i=0}^{n-1} (x-x_i) \quad . \tag{4.16}$$

Now $|p(x)|_{m_i(x)} = p(x_i)$ for any $p(x) \in F[x]$, and $s_i^{(k)} = 1/(x_k - x_i)$ from (4.13). Thus, if we define

$$r_i^{(k)} = 1/s_i^{(k)} = x_k - x_i \quad , \tag{4.17}$$

$$b_k = 1/c_k = \prod_{i=0}^{k-1} (x_k - x_i) \qquad \text{(cf. (3.13))} \quad ,$$

then (3.12), which serves as the basis for Algorithm N1, becomes

$$a_k = (u_k - ((\ldots(a_{k-1}r_{k-2}^{(k)} + a_{k-2})r_{k-3}^{(k)} + \ldots)r_0^{(k)} + a_0))/b_k . \tag{4.18}$$

Similarly, (3.14), which serves as the basis for Algorithm N2, becomes

$$a_k = (\ldots((u_k - a_0)/r_0^{(k)} - a_1)/r_1^{(k)} - \ldots - a_{k-1})/r_{k-1}^{(k)} \quad , \tag{4.19}$$

and (3.6) and (3.7'), which serve as the basis for Algorithm N3, become

$$u^{[k]}(x) = u^{[k-1]}(x) + a_k \prod_{i=0}^{k-1} (x-x_i) \tag{4.20}$$

and

$$a_k = (u_k - u^{[k-1]}(x_k))/b_k \quad . \tag{4.21}$$

On substituting $1/x_k - x_i$ for $s_i^{(k)}$ in (3.15 - 3.15") we obtain

$$[x-x_k] = u(x_k) \quad , \tag{4.22}$$

$$[x-x_0, x-x_k] = \frac{[x-x_k] - [x-x_0]}{x_k - x_0} \quad , \tag{4.22'}$$

$$[x-x_0, \ldots, x-x_{i-1}, x-x_k]$$

$$= \frac{[x-x_0, \ldots, x-x_{i-2}, x-x_k] - [x-x_0, \ldots, x-x_{i-2}, x-x_{i-1}]}{x_k - x_{i-1}} \tag{4.22''}$$

We observe that the multiplied-differences introduced in the context of an arbitrary Euclidean domain have specialized in the polynomial case to the divided-differences of the classical Newton's Interpolation Polynomial, the latter being given by (4.16) (thus the term "Newtonian" that we have employed in the more general algebraic contexts of Sections 2 and 3). The customary notation for the divided-difference $[x-x_0, x-x_1, \ldots, x-x_{i-1}, x-x_k]$ is $[x_0, x_1, \ldots, x_{i-1}, x_k]$.

We apply the above Newton polynomial formulas in

Example 4.4. Compute the solution u(x) to the Chinese Remainder (Interpolation) Problem of Example 4.2 (u(x) has the modular representation [10, 334, 1040, 5920] with respect to the moduli $\underline{m}(x)$ = (x-1, x-4, x-6, x-11)), this time employing Newtonian methods.

First we consider the (precomputed) constants needed by the Newtonian Algorithms N1, N2, and N3. For the case of the Interpolation Problem in $\underline{Z}[x]$, it is more convenient to have $r_i^{(k)} = 1/s_i^{(k)}$, $b_k = 1/c_k$ of (4.17) (integer quantities), rather than the $s_i^{(k)}$'s and c_k's themselves (rational quantities). Thus, for example, lines 7 and 8 of Algorithm N1 would utilize these constants according to

$$t := t \times r_i^{(k)} + a_i \; ;$$

$$a_k := (u_k - t)/b_k \; ;$$

and similarly for Algorithms N2 and N3.

For the problem under consideration, the constants $r_i^{(k)}$, b_k, and $q_k(x)$ (cf. (3.16)) are as follows:

$$r_0^{(1)} = 3, \quad r_0^{(2)} = 5, \quad r_0^{(3)} = 10$$

$$r_1^{(2)} = 2, \quad r_1^{(3)} = 7$$

$$r_2^{(3)} = 5$$

$$b_1 = 3, \quad b_2 = 10, \quad b_3 = 350$$

$$q_1(x) = x - 1$$

$$q_2(x) = x^2 - 5x + 4$$

$$q_3(x) = x^3 - 11x^2 + 34x - 24 .$$

Method 1: Following Algorithm N1 and (4.18),

$$a_0 = 10$$

$$a_1 = (334-10)/3 = 108$$

$$a_2 = (1040 - (108\times5 + 10))/10 = 49$$

$$a_3 = (5920 - ((49\times7 + 108)10 + 10)/350 = 4 ,$$

whence from (4.16) and (3.17)

$$\begin{aligned} u(x) &= 10 + 108(x-1) + 49(x-1)(x-4) + 4(x-1)(x-4)(x-6) \\ &= (((4(x-6) + 49)(x-4) + 108)(x-1) + 108) + 10 \\ &= 4x^3 + 5x^2 - x + 2 . \end{aligned}$$

Method 2: Following Algorithms N2 and (4.19),

$$a_0 = 10$$

$$a_1 = (334-10)/3 = 108$$

$$a_2 = ((1040-10)/5 - 108)/2 = 49$$

$$a_3 = (((5920-10)/10 - 108)/7 - 49)/5 = 4$$

Conversion to standard form is carried out as in Method 1.

The intermediate values in the above computation of the a_k's constitute the following table of divided-differences (analogous to the table of multiplied-differences in the integer case of Example 4.3).

Table 4.2. Divided-Differences of [10, 334, 1040, 5920] with Respect to m(x) = (x-1, x-4, x-6, x-11)

[ ]	[,]	[, ,]	[, , ,]
$10 = a_0$			
334	$108 = a_1$		
1040	206	$49 = a_2$	
5920	591	69	$4 = a_3$

Method 3: Following Algorithm N3 and (4.21), v, a, U are successively computed as

k=0: $U(x) = 10$

k=1: $v = U(4) = 10$

$a = (334-10)/3 = 108$

$U(x) = 10 + 108(x-1)$

$= 108x-98$

k=2: $v = U(6) = 550$

$a = (1040-550)/10 = 49$

$U(x) = 108x-98 + 49(x^2-5x+4)$

$= 49x^2-137x+98$

k=3: $v = U(11) = 4520$

$a = (5920-4520)/350 = 4$

$U(x) = 49x^2-137x+98 + 4(x^3-11x^2+34x-24)$

$= 4x^3+5x^2-x+2$

The final value of U(x) is then the solution to the given Chinese Remainder (Interpolation) Problem, agreeing with the solution obtained by the other Newtonian and Lagrangian methods.

We return now to Algorithm N2 (which employs divided-differences either implicitly or explicitly), and observe a noteworthy aspect of Table 4.2--the entries are all integral, which means that the indicated divisions in (4.22' - 4.22") have been *exact*. This is a most important property of divided-difference tables associated with interpolation problems in $\underline{Z}[x]$ if it should be the case in all such problems, for it means that all the values (both intermediate and final) generated by the Newtonian Algorithm N2 (and hence Algorithms N1 and N3) are integer, which in turn means that *no rational mode arithmetic is required*--a most important implementation consideration.

We now prove this important "fraction-free" property under the more general conditions of

Theorem 4.1. Consider the Interpolation Problem in D[x], where D is an integral domain. (Thus the solution polynomial $u(x) \in D[x]$ and the evaluation points x_i $(i=0,1,\ldots,n)$ are chosen in D--see the discussion "One remark on terminology" following (4.8).) Then the divided differences (4.22 - 4.22") are all in D (i.e., the indicated divisions are all exact so that no quotients are produced).

Proof. Define the sequence $d_0(x), d_1(x), \ldots, d_n(x) \in Q(D)[x]$ according to

$$d_0(x) = u(x) \tag{4.23}$$

$$d_1(x) = \frac{d_0(x) - a_0}{x - x_0} \tag{4.23'}$$

and recursively

$$d_i(x) = \frac{d_{i-1}(x) - a_{i-1}}{x - x_{i-1}} \tag{4.23"}$$

It follows from the divided-difference recursion (4.19 - 4.19") (and the fact that $a_i = [x-x_0,\ldots,x-x_{i-2},x-x_{i-1}]$ by the Corollary to Theorem 3.2) that

$$d_i(x_k) = [x-x_0,\ldots,x-x_{i-1},x-x_k] \qquad (i\leq k\leq n) \tag{4.24}$$

and, in particular,

$$d_i(x_i) = a_i \quad . \tag{4.25}$$

It follows from (4.24) that if we can show that each $d_i(x) \in D[x]$, then the theorem is established.

Now $d_0(x) = u(x) \in D[x]$, by assumption. Proceeding by induction, assume $d_{i-1}(x) \in D[x]$. By the (limited) Division Algorithm in D[x] (see [1, p. 39]),

$$d_{i-1}(x) = (x-x_{i-1})q(x) + r \quad , \tag{4.26}$$

where $q(x) \in D[x]$, $r \in D$. Setting $x = x_{i-1}$ yields $r = d_{i-1}(x_{i-1}) = a_{i-1}$ by (4.25). Then

$$q(x) = \frac{d_{i-1}(x) - a_{i-1}}{x-x_i}$$

$$= d_i(x) \text{ by (4.23")} \quad ,$$

so that $d_i(x) \in D[x]$, completing the proof of the above assertion, and thence the proof of Theorem 4.1.

The More General Chinese Remainder Problem in F[x]--an Example

Although in F[x] we have mainly restricted our attention to a special (but most important) case of the Chinese Remainder Problem, namely the polynomial Interpolation Problem where the moduli polynomials are all linear, the various solutions that have been presented are by no means restricted to this special case. In order to illustrate the more general nature of the Chinese Remainder Problem in F[x], we conclude this section with

Example 4.5. u(x) has the modular representation $[-8, 11x+17, 17x^2-x-2]$ with respect to the moduli $\underline{m}(x) = (x+2, x^2-3, x^3-3x^2+1)$. (Thus $u(x) \equiv -8 \pmod{x+2}$, etc.) Compute u(x).

Solution. First we note that u(x) is determined uniquely modulo $m_0(x)m_1(x)m_2(x)$, whence there is a unique polynomial with degree ≤ 6 having the above modular representation: this polynomial is then *the* solution u(x).

We proceed by the Newtonian Formula (3.8), which becomes

$$u(x) = a_0 + a_1(x+2) + a_2(x+2)(x^2-3) \quad .$$

The a_i's are determined by computing the table of multiplied-differences (which can be regarded in this context as "generalized" divided-differences) according to (3.15 - 3.15"). (Thus we are essentially using (3.14) and Algorithm N2.) First we need the $s_i^{(k)}$'s of (3.3):

$$s_0^{(1)} = -x+2$$

$$s_0^{(2)} = \frac{x^2-5x+10}{19} \quad , \quad s_1^{(2)} = \frac{3x^2-x-15}{37} \quad ,$$

and the following table is computed.

Table 4.3. Divided-Differences of $[-8, 11x+17, 17x^2-x-2]$ with Respect to $\underline{m}(x) = (x+2, 11x+17, 17x^2-x-2)$

$[\ ]$	$[\ ,\ ]$	$[\ ,\ ,\ ]$
$-8 = a_0$		
$11x+17$	$-3x+17 = a_1$	
$17x^2-x-2$	$4x^2-3x+5$	$4 = a_2$

The desired solution is then

$$u(x) = -8 + (-3x+17)(x+2) + 4(x+2)(x^2-3)$$
$$- 4x^3 + 5x^2 - x + 2 .$$

5. ANALYSIS AND APPRAISAL OF CHINESE REMAINDER AND INTERPOLATION ALGORITHMS

In the previous sections we have identified and illustrated two classes of methods, Lagrangian and Newtonian, for computing the solution to the Chinese Remainder Problem in increasingly specialized contexts. Because of the general validity (and hence interchangeability) of these methods, it is appropriate to analyze the various algorithms based on the Lagrangian and Newtonian Formulas with respect to time and storage requirements in an effort to answer the important question "which one is best?" This section is devoted to such an analysis and appraisal, carried out in the contexts of (i) the Chinese Remainder Problem in $\underline{Z}$, and (ii) the Interpolation Problem in $\underline{Z}[x]$--the two cases of primary importance for symbolic mathematics by computer. The assumptions made in the analysis of algorithms are especially relevant to their application to exact and symbolic computation, as shall be explained.

In $\underline{Z}$ the Chinese Remainder Problem is: Compute the solution u $(0 \leq u < \prod_i m_i)$ to the system of congruences

$$u \equiv u_i \pmod{m_i} \qquad (i=0,1,\ldots,n) \quad . \tag{5.1}$$

It is assumed that the moduli m_i are specified and *fixed* beforehand, so that precomputation involving these m_i's may be carried out. Also, it is assumed that both the m_i's and the u_i's are single-precision integers.

These assumptions reflect the situation wherein residue methods are employed to obtain the "exact" solution (i.e., the solution over the rationals $\underline{Q}$ instead of over the reals $\underline{R}$, thus circumventing round-off error) to a variety of computational problems, such as matrix inversion and characteristic polynomial evaluation. The basic idea* here is first to transform the problem so that the solution can be expressed in terms of integers (eg. over $\underline{Q}$ to determine the numerators and denominators of rational numbers), and then to carry out the computations over the field

---

*The sketchy description of the residue method given here is sufficient for our motivational purposes. Complete references to the literature in this area are given in the next section.

$\underline{Z}_p$ of integers modulo p for *several* primes p in order to eventually reconstruct the integer solution by application of a Chinese Remainder formula. In such applications the moduli m_i are typically chosen as the n+1 largest primes that fit into a machine word.

Finally we make the following assumptions concerning multiple-precision integer arithmetic, which are consistent with the standard algorithms and machine hardware employed for this purpose. If a and b are integers with precision pr(a) ≃ n and pr(b) ≃ m ($n \leq m$), then the number of single-precision (machine) additions ($\#_+$), multiplications ($\#_\times$), and divisions ($\#_\div$) required for computing $a \pm b$, $a \times b$, $a \div b$ are given by

$$
\begin{aligned}
a \pm b: \quad & \#_+ \simeq m \\
a \times b: \quad & \#_+ \simeq 2mn-m-n \\
& \#_\times \simeq mn \\
a \div b: \quad & \#_+ \simeq (2n-1)(m-n) \\
& \#_\times \simeq (m-n)n \\
& \#_\div \simeq m-n
\end{aligned}
\tag{5.2}
$$

with the precision of the results being given by

$$
\begin{aligned}
& pr(a \pm b) \simeq m \\
& pr(a \times b) \simeq m+n \\
& pr(r_a(b)) \simeq n \qquad (r_a(b) = \text{remainder when b is divided by a})
\end{aligned}
\tag{5.3}
$$

The number of words of memory required to store a,b,...,c is denoted by st(a,b,...,c).

We now proceed with the analyses of the various *integer* Chinese Remainder Algorithms.

Integer Lagrangian Chinese Remainder Algorithm: The solution $u \in \underline{Z}$ to (5.1) is computed according to (3.4 - 3.5). The Lagrangian coefficients

L_k of (3.4) are precomputed and stored. The number* of machine operations and words of storage (for precomputed values) are given according to (5.2 - 5.3) by

$$\#_+ \simeq 2n^2$$
$$\#_\times \simeq n^2 \qquad (5.4)$$

with

$$st(L_0,L_1,\ldots,L_n) \simeq n^2 \ .$$

The operations $(+,\times,\div)$ required for the multiple-precision divisions of u obtained by (3.5) ($pr(u) \simeq n+2$) by $M = \prod_i m_i$ ($pr(M) \simeq n+1$) in reducing u to the range $0 \leq u < M$ have been ignored, since they do not contribute to the order n^2 terms of (5.4).

Integer Newtonian Chinese Remainder Algorithms: We analyze first the Newtonian Algorithms N1 and N2, then N3.

It is convenient in the analysis (and application) of Algorithm N1 or N2 to regard the solution u to the Chinese Remainder Problem (5.1) as being computed in two stages: first is the conversion from modular (3.9) to Newton (3.10) representation according to Algorithm N1 or N2; second is the conversion from Newton (mixed-radix) to standard (fixed-radix) representation via Horner's scheme (3.17). In particular, the operations of this second stage are included in the analysis of Algorithms N1 and N2.

In the Newtonian Algorithms it is assumed that the operations enclosed within the modulo m operator $|\ |_m$ are *all* carried out modulo m; thus, for example, $|a + b\times c|_m$ is computed as $|a + |b\times c|_m|_m$. Also it is assumed that each modulo m operation is carried out by the corresponding integer operation followed by a division by m in order to render the result in the range $0 \leq x < m$.

---

*When the number of operations and words of storage are given by polynomial functions of n, only the highest-order term in each case is retained. For moderate and large values of n (the cases of interest for computing), the time and storage requirements are essentially determined by these highest-order terms.

With the constants c_k of (3.13) and $s_i^{(k)}$ of (3.3) precomputed for use in Algorithms N1 and N2 respectively, the operations counts are given by

Stage 1: (Modular to Newton conversion using Algorithm N1 or N2) (5.5)

$$\#_+ \simeq n^2/2$$

$$\#_\times \simeq n^2/2$$

$$\#_\div \simeq n^2$$

Stage 2: (Newton to fixed-radix conversion using (3.17))

$$\#_+ \simeq n^2/2$$

$$\#_\times \simeq n^2/2$$

Combining the two stages gives the overall operations counts

$$\#_+ \simeq n^2 \qquad (5.6)$$

$$\#_\times \simeq n^2$$

$$\#_\div \simeq n^2$$

with storage requirements

$$st(c_1,\ldots,c_n) \simeq n \text{ for Algorithm N1} \qquad (5.7)$$

$$st(s_0^{(1)},\ldots,s_{n-1}^{(n)}) \simeq n^2/2 \text{ for Algorithm N2.}$$

Turning to the analysis of Algorithm N3, at step k of the iteration we have by (5.2) that $pr(U = u^{[k-1]}) \simeq k$. By (5.1) the number of operations at step k required to compute v is $\#_+ \simeq k-1$, $\#_\times \simeq k-1$, $\#_\div \simeq k-1$; to compute a is $\#_+ = 1$, $\#_\times = 1$, $\#_\div = 2$; and to compute $U = u^{[k]}$ is $\#_+ \simeq 2k-1$, $\#_\times \simeq k$. The overall operations counts for Algorithm N3 are then given by

$$\#_{+} \simeq 3n^2/2 \qquad (5.8)$$

$$\#_{\times} \simeq n^2$$

$$\#_{\div} \simeq n^2/2$$

with storage requirements (taking into account that $pr(q_k) \simeq k$)

$$st(c_1,\ldots,c_n;q_1,\ldots,q_n) \simeq n^2/2 \quad . \qquad (5.9)$$

The results of the analyses of the various integer Chinese Remainder Algorithms are summarized in the following table. ($\#_T$ = Total number of machine operations $+,\times,\div$.)

Table 5.1. Operations and Storage Requirements for Integer Chinese Remainder Algorithms

Algorithm	$\#_+$	$\#_\times$	$\#_\div$	$\#_T$	storage
Lagrangian	$2n^2$	n^2		$3n^2$	n^2
Newtonian - N1	n^2	n^2	n^2	$3n^2$	n
- N2	n^2	n^2	n^2	$3n^2$	$n^2/2$
- N3	$3n^2/2$	n^2	$n^2/2$	$3n^3$	$n^2/2$

We now analyze the various *polynomial* Chinese Remainder (Interpolation) Algorithms. Here we are concerned specifically with the Interpolation Problem in $\underline{Z}[x]$: Compute $u(x) \in \underline{Z}[x]$* such that

$$u(x_k) = u_k \qquad (k=0,1,\ldots,n) \quad , \qquad (5.10)$$

where $x_k, u_k \in \underline{Z}$, and deg $u(x) \leq n$.

---

*Cf. the discussion "Remark on terminology" following (4.8) for the motivation underlying the assumption that the interpolation polynomial $u(x) \in \underline{Z}[x]$ rather than $\underline{Q}[x]$.

Polynomial Lagrangian Interpolation Algorithm. The solution $u(x) \in \underline{Z}[x]$ to (5.10) is computed according to the following embellished version of (4.14 - 4.15); the embellishments are for the purpose of avoiding intermediate rational mode arithmetic that would be required if (4.14 - 4.15) were applied explicitly (cf. Example 4.2).

From (4.14) we have

$$L_k(x) = \ell_k(x)/d_k \quad , \tag{5.11}$$

where

$$\ell_k(x) = \prod_{i \neq k} (x-x_i) \in \underline{Z}[x], \quad d_k = \prod_{i \neq k} (x_k-x_i) \in \underline{Z} \quad .$$

Now define

$$\alpha = \text{g.c.d.}\ (d_0,d_1,\ldots,d_n) \tag{5.12}$$

$$\beta = \prod_i d_i/\alpha \quad (= \text{l.c.m.}\ (d_0,d_1,\ldots,d_n))$$

$$\gamma_k = \prod_{i \neq k} d_i/\alpha \ .$$

Then $u(x)$ is computed according to the *fraction-free*

$$u(x) = \frac{\sum_k u_k \gamma_k \ell_k(x)}{\beta} \quad , \tag{5.13}$$

which clearly yields the same final result as (4.15).

With $\ell_k(x)$, γ_k, and β precomputed, the integer operations counts (ignoring those of order n) are given by

$$\begin{aligned} \#_+ &\simeq n^2 \\ \#_\times &\simeq n^2 \end{aligned} \tag{5.14}$$

with storage requirements

$$\text{st}(\ell_0(x),\ldots,\ell_n(x);\gamma_0,\gamma_1,\ldots,\gamma_n;\beta) \simeq n^2 \quad .$$

Polynomial Newtonian Chinese Remainder Algorithms: We analyze the Newtonian Algorithms N1, N2, and N3 in turn.

As the integer case, it is convenient in the analysis (and application) of Algorithm N1 or N2 to consider the solution $u(x)$ to the Interpolation Problem (5.10) as being computed in two stages. First is the conversion from modular (sample-value) representation $[u_0,u_1,\ldots,u_n]$ to the Newton representation $<a_0,a_1,\ldots,a_n>$ corresponding to (4.16); second is the conversion from the Newton representation to standard representation $u(x) = s_n x^n+s_{n-1}x^{n-1}+\ldots+s_0$ by applying Horner's scheme (3.17) to (4.16), which becomes

$$u(x) = (\ldots(a_n(x-x_{n-1}) + a_{n-1})(x-x_{n-2})+\ldots+a_1(x-x_0) + a_0 \ . \quad (5.16)$$

The operations counts (ignoring those of order n) for Algorithm N1 (following (4.18)) are then given by

Stage 1: (Sample-value to Newton conversion - Algorithm N1 proper) (5.17)

$$\#_+ \simeq n^2/2$$

$$\#_\times \simeq n^2/2$$

Stage 2: (Newton to standard form conversion using (5.14))

$$\#_+ \simeq n^2/2$$

$$\#_\times \simeq n^2/2$$

Combining the counts for two stages gives

$$\#_+ \simeq n^2 \quad (5.18)$$

$$\#_\times \simeq n^2$$

with storage requirements

$$st(r_0^{(1)},\ldots,r_{n-1}^{(n)};b_k) \simeq n^2 . \quad * \tag{5.19}$$

The operations counts for Algorithm N2 (following (4.19)) are the same as for Algorithm N1 except that the multiplications of stage 1 are replaced by divisions. Thus the overall operations counts corresponding to (5.18) is

$$\begin{aligned} \#_+ &\simeq n^2 \\ \#_\times &\simeq n^2/2 \\ \#_\div &\simeq n^2/2 \end{aligned} \tag{5.20}$$

with storage requirements

$$st(r_0^{(1)},\ldots,r_{n-1}^{(n)}) \simeq n^2 \quad . \tag{5.21}$$

As for Algorithm N3 (see also (4.20 - 4.21)), at step k of the iteration the polynomial $U(x) = u^{[k-1]}(x)$ has degree k-1. Thus, the number of operations at step k required to compute $v = U(x_k)$ is (applying Horner's Rule) $\#_+ = k-1$, $\#_\times = k-1$; to compute a is $\#_+ = 1$, $\#_\div = 1$; and to compute $U(x) = u^{[k]}(x)$ (assuming the $q_k(x) = \prod_{i=0}^{k-1}(x-x_i)$ are precomputed) is $\#_+ = k-1$, $\#_\times = k-1$. The overall operations counts for Algorithm N3 are then given by

$$\begin{aligned} \#_+ &\simeq n^2 \\ \#_\times &\simeq n^2 \end{aligned} \tag{5.22}$$

with storage requirements (taking into account that deg $q_k(x) = k$)

$$st(q_1(x),\ldots,q_n(x);b_1,\ldots,b_n) \simeq \frac{n^2}{2} \quad . \tag{5.23}$$

---

*Of course the storage requirements can be reduced to $\simeq n$ by computing the simple quantities $r_i^{(k)} = x_k - x_i$. We have adopted the policy of precomputing all precomputable constants not so much for efficiency as for uniformity.

The results of the analyses of the various polynomial Interpolation Algorithms are summarized in the following table.

Table 5.2. Operations and Storage Requirements for Polynomial (Z[x]) Chinese Remainder Algorithms

Algorithm	$\#_+$	$\#_\times$	$\#_\div$	$\#_T$	storage
Lagrangian	n^2	n^2		$2n^2$	n^2
Newtonian - N1	n^2	n^2		$2n^2$	n^2
- N2	n^2	$n^2/2$	$n^2/2$	$2n^2$	n^2
- N3	n^2	n^2		$2n^2$	n^2

Note that in the analysis of the polynomial Chinese Remainder Algorithms, we have made the tacit assumption that precomputed and computed integer quantities (including polynomial coefficients) are all single-precision. This may not be a realistic assumption when the solution polynomial has large (albeit single-precision) coefficients, or when the solution polynomial has very small (say single digit!) coefficients but moderate or large degree; e.g., consider the evaluation of $u(x) = 2^{30}x^3$ for $x = 2$, or $u(x) = x^{15}$ for $x = 10$, on a machine with a 32-bit word. Also in the Lagrangian case the precomputed quantity β in (5.12) will typically be multiple-precision, necessitating n multiple-precision divisions in (5.13).

In order to avoid the problem of having integer overflow preventing the determination of single-precision results, the various polynomial Chinese Remainder Algorithms can be carried out using modulo p arithmetic, where p is chosen as the largest single-precision prime. Thus we compute the solution u(x) to the Interpolation Problem (5.10) in $\underline{Z}_p[x]$ (where the u_k's are reduced modulo p) instead of in $\underline{Z}[x]$. Here we are exploiting the simple fact that if an integer x (specifically a coefficient of the solution polynomial u(x)) satisfies

$$|s| \leq (p-1)/2 \quad , \tag{5.24}$$

then $s = |s|_p$ provided only that the modulo p operator is (re-)defined to return its value in the range $\pm (p-1)/2$.

Carrying out the operations of (4.14 - 4.15) (Lagrangian Algorithm), (4.18), (4.19), (4.20 - 4.21) (Newtonian Algorithms N1, N2, N3, respectively) modulo p--in particular all indicated divisions become modulo p multiplications by (precomputed) modulo p inverses--it is readily verified that the four algorithms have the same (approximate) operations counts and storage requirements, given by

$$\begin{aligned} \#_{+} &\simeq n^2 \\ \#_{\times} &\simeq n^2 \\ \#_{\div} &\simeq 2n^2 \end{aligned} \tag{5.25}$$

with

$$\text{storage} \simeq n^2 \quad .$$

Finally, we note that there is one possible pitfall to this modulo p approach to the Interpolation Problem in $\underline{Z}[x]$, namely one or more of the coefficients of the solution polynomial u(x) may be beyond the range (5.24). But then we can solve the $\underline{Z}_p[x]$ Interpolation Problem for *several* primes p_i in order to reconstruct $u(x) \in \underline{Z}[x]$ by applying one of our *integer* Chinese Remainder Algorithms to the modulo p_i values of the coefficients in turn. Moreover, the process just described is the basis of a recursive solution to the *multi-variate* Interpolation Problem: Determine the solution $u(x,y,\ldots,z) \in \underline{Z}[x,y,\ldots,z]$ satisfying

$$u(x_i,y_j,\ldots,z_k) \equiv u_{i,j,\ldots,k} \pmod{p_\ell} \tag{5.26}$$

$$(i=0,1,\ldots,n_x;\ j=0,1,\ldots,n_y;\ \ldots;\ k=0,1,\ldots,n_z;\ \ell=0,1,\ldots,n_p) \ .$$

The solution $u(x,y,\ldots,z)$ is obtained by interpolating in turn with respect to the moduli $z-z_k,\ldots,y-y_j,x-x_i$ and finally solving integer Chinese Remainder Problems with respect to the moduli p_ℓ in order to recover the integer coefficients of the solution polynomial $u(x,y,\ldots,z)$, which is of

degree $\leq n_x$ in x, $\leq n_y$ in y ,..., $\leq n_z$ in z. We shall not discuss the implementation details of this multi-variate interpolation process here.

Appraisal of the Chinese Remainder Algorithms. The operations counts in Tables 5.1 and 5.2 show that the variations in execution time among the Lagrangian and Newtonian Algorithms are not going to be significant. Thus, other considerations are more decisive in choosing a method; we will now discuss these.

The integer and polynomial algorithms based on the Lagrangian Formula (3.4 - 3.5) have one bad property, in that the moduli m_i must all be known in advance in order to compute the Lagrangian coefficients (3.4). This is a serious disadvantage in practice, for even when the successive moduli $m_0, m_1, \ldots$ are known and fixed, it is frequently the case that the *number* of moduli that must be used is not known in advance but depends (at execution time) on the larger problem at hand.

On the other hand, the Newtonian Algorithms N1, N2, N3 based on (3.6 - 3.8) are very favorable in this regard: increasing the number of moduli from n to n+1 (at execution time) simply entails performing the basic iteration one additional time in generating a_{n+1} of the Newtonian representation (3.10) (Algorithm N1, N2) or in passing from $U = u^{[n]}$ to $U = u^{[n+1]}$ (Algorithm N3). This very important property of the Newtonian Algorithms N1, N2, and N3, wherein the addition of a modulus is accomplished by carrying out an additional iteration of the basic scheme (and utilizing the previously generated information), we call *extensibility*.

We thus restrict our attention to the Newtonian Algorithms. Using Algorithm N1 or N2, the solution u to the integer Chinese Remainder Problem (5.1) is computed in two stages (see (5.5)). Stage 1 (Algorithm N1 or N2 proper) involves only single-precision calculation; all multiple-precision calculations are carried out in stage 2 (the conversion from Newton (mixed-radix) to fixed-radix form by (3.17)). This division of labor into these two stages is propitious, for in some applications the Newton representation of the solution u may be adequate; e.g., in comparing the magnitude of two quantities u, u'. In such applications, stage 2 with its attendant multiple-precision arithmetic need not be carried out. In such cases any one of the standard algebraic compiler languages such as Fortran,

Algol, or PL/I (none of which have a multiple-precision integer arithmetic capability) is completely adequate for the implementation.

In choosing between Algorithms N1 and N2, the former is to be preferred because of the much smaller storage requirements for precomputed quantities (see Table 5.1). (Indeed, Algorithm N2 was considered only because of its relationship in the polynomial case to the divided differences of Newton's Interpolation Polynomial.)

As for Algorithm N3, it requires multiple-precision calculation throughout its basic iteration. For that reason and because its storage requirements are much greater than those of Algorithm N1, we choose the latter. However, from Table 5.1 we observe that Algorithm N3, while having (approximately) the same number of machine operation as Algorithm N1, does have fewer divisions and more additions. Thus if division is very slow relative to addition, one might choose Algorithm N3 over N1.

In the polynomial case, Table 5.1 indicates there is little to differentiate with respect to time and storage between Algorithms N1 and N3. However, as in the integer case, Algorithm N1 offers the flexibility of stopping after stage 1 when the Newton polynomial (4.16) has been computed, and thus is to be preferred over N3.

Thus we can conclude this appraisal of our various Chinese Remainder Algorithms with the recommendation that the Newtonian Algorithm N1 be used for either the integer or polynomial Chinese Remainder Problem.

6. HISTORICAL, BIBLIOGRAPHICAL, AND CONCLUDING REMARKS

In [2, p. 407], E. T. Bell states, "The history of interpolation formulae is complicated and controversial." Certainly the same statement applies to the history of the (integer) Chinese Remainder Problem. In this section we first attempt to briefly delineate the interesting history of Chinese Remainder and Interpolation methods, with emphasis on those aspects that are relevant to computation.

The origins of the Chinese Remainder Problem are not at all certain, with dates ranging from 200 B.C. to 400 A.D. Dickson [6, p. 57] attributes the solution (in the form of an obscure verse!) of a very special case of the Chinese Remainder Problem to the Chinese Mathematician Sun-Tsu around the first century A.D. This solution was essentially Lagrangian (i.e., according to (3.4 - 3.5)) in character.

Although Dickson [6, pp. 57-64] mentions the work of some fifty mathematicians (including Euler) in connection with the Chinese Remainder Problem, there is little doubt that the most comprehensive and algorithmic treatment of this problem was due to Gauss in his celebrated "Disquisitiones Arithmeticae" [8] (which has recently been translated into English).

In [8, Art. 32], Gauss has presented a Newtonian* method for solving the Chinese Remainder Problem.** The method described is essentially that of Algorithm N2 or N3, but the level of detail does not allow us to decide which. However, quite evident in the discussion of Art. 32 is the Newton (mixed-radix) Formula (3.8) for the solution.

In [8, Art. 36], Gauss has given what is essentially the Lagrangian solution (3.4 - 3.5) of this paper. (His stated preference for the Lagrangian over the Newtonian method of solution can perhaps be challenged in view of the results of Sec. 5 of this paper.)

---

*The terms "Newtonian" and "Lagrangian" in connection with solutions to the integer Chinese Remainder Problem are, of course, the author's own terminology.

**Actually Gauss considers the more general Chinese Remainder Problem wherein the moduli are not necessarily relatively prime. This generalization is not of practical interest for us, because in applications to computation the moduli m_i are typically chosen not only as relatively prime but as prime so that the integers modulo each m_i have the structure of a field.

It is interesting to note that although Gauss and Euler were great computers (in the personal sense of that word) there is no indication in their works on the Chinese Remainder Problem that they considered the solu- to be of any computational value, such as in calculating with large integers--to them as to others the Chinese Remainder Problem was a puzzle from Number Theory.

The first computational application of the Chinese Remainder Problem seems to have occurred as late as the mid 1950's when two Czechoslovakian computer designers, A. Svoboda and M. Valach, realized the advantages of machines whose arithmetic was carried out in a modular fashion (for several "hard-wired" moduli), the foremost of these advantages being the carry-free nature of addition and multiplication. The Chinese Remainder Problem in this case was essentially the problem of conversion of numbers from internal (modular) form to external (standard decimal integer) form. Independently and slightly later, these same ideas occurred to H. Aiken and H. Garner in the United States.

It is not our purpose to delineate the tremendous amount of research conducted in the area of modular arithmetic computers. An extensive account and annotated bibliography is given by Szabo and Tanaka [17]. The prevailing opinion concerning the modular arithmetic approach to general-purpose computer hardware design is that the experiment was largely a failure. Although modular arithmetic computers afforded a few advantages over traditional design, it turned out that certain basic operations such as sign detection and scaling (which are trivial using standard representations for numbers) are *intrinsically* difficult when numbers are expressed in modular form, i.e., no efficient algorithms *exist* for these operations--see [17, Sec. 4.3] for such a result.

Much more successful have been the programming (i.e., software as opposed to hardware) applications of modular arithmetic, described in Takahasi and Ishibashi [18], Borosh and Fraenkel [3], and Newman [14]. These papers all investigated the exact solution of linear equations using modular arithmetic along the lines sketched near the beginning of Sec. 5. They employed a variety of Chinese Remainder Formulas (both Lagrangian and Newtonian) in their solutions. Recently Cabay [4] has considered the same problem, suggesting several points of improvement over previous methods.

The theory and practice of polynomial interpolation has for its starting point the definitive and celebrated work of Newton, specifically Lemma 5, Book 3 of the *Principia* [15], where Newton's Interpolation Formula and associated table of divided differences (essentially Algorithm N2 of this paper) first appeared.*

Lagrange's Interpolation Formula was given by Lagrange in [12]. However, this formula was discovered earlier by Waring [19].**

In [9], Gauss discussed both Lagrange's and Newton's Interpolation Formula. It is thus noteworthy that Gauss considered in depth both cases of the Chinese Remainder Problem, integer and polynomial (and for each case considered both methods of solution, Lagrangian and Newtonian), yet he did not seem to recognize any relationship between the two cases. Perhaps this stems from the fact that the algorithmic details and contexts were quite different for the two problems--Gauss was very much the pure mathematician (number theorist) in the integer case, and applied mathematician (astronomer) in the polynomial case.

Of course, the standard applications of polynomial interpolation formula are numerical--the interpolation polynomial is not just constructed but also evaluated for intermediate values of the argument. Indeed, Newton applied his interpolation formula in [15, p. 500] to a problem from astronomy: "Certain observed places of a comet being given, to find the place of the same at any intermediate given time."

The idea of constructing the interpolation polynomial as a symbolic entity (i.e., for other than interpolatory purposes) can be found in the numerical analysis literature, specifically the method for computing the

---

*Newton's own comments on the Interpolation Problem and his solution are most revealing as to his own feeling of its merit [10, p. 45]: "To describe a geometrical curve which will pass through any given points... although the problem may seem intractable at first sight it is nevertheless the contrary. Perhaps it is indeed one of the prettiest problems that I can hope to solve."

**It seems strange that Waring's paper should have remained unacclaimed, especially in view of its clarity and prominent place of publication.

coefficients of the characteristic polynomial $c(\lambda) = |\lambda I - A|$ by constructing the interpolation polynomial passing through $c(\lambda_i)$ for arbitrary distinct choices $\lambda_1,\ldots,\lambda_n$ of the eigenvalue parameter λ.

Application of interpolation methods for symbolic computation has been suggested by Takahasi and Ishibashi in [18, Sec. 4], and anticipated by Collins in [5]. An especially interesting application of exact interpolation methods occurs in the fact (multiple-precision) multiplication scheme due to Toom and Cook (see [11, Sec. 4.3.3]).

The Chinese Remainder Theorem has been presented in several places at various levels of abstraction (e.g. cf. [13; pp. 161-2, p. 165 Pr. 5]). Thus, the main result of Sec. 2 is not the Chinese Remainder Theorem per se but the proof presented there which explicitly establishes two Chinese Remainder Formulas (Lagrangian and Newtonian) of distinct computational character, and does so in a context sufficiently abstract to make evident the underlying algebraic nature of the Chinese Remainder Problem and methods of solution.

Likewise, various Chinese Remainder and Interpolation schemes have been described (e.g. cf. [11; p. 254, pp. 428-30]). The main purpose of Secs. 3-5 was the systematic derivation and analysis of the naturally suggested algorithms, in contexts relevant to symbolic computation.

7. REFERENCES

1. Albert, A. A., *Fundamental Concepts of Algebra*, University of Toronto Press, Toronto, 1956.

2. Bell, E. T., *The Development of Mathematics*, McGraw-Hill Book Co., New York, 1945.

3. Borosh, I. and A. S. Fraenkel, "Exact Solution of Linear Equations with Rational Coefficients by Congruence Techniques," *Mathematics of Computation*, Vol. 20, No. 93 (Jan. 1966), pp. 107-112.

4. Cabay, S., "Exact Solution of Linear Equations," *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, March, 1971.

5. Collins, G. E., et al., *The SAC-1 Modular Arithmetic System*, University of Wisconsin Computing Center, Technical Reference No. 10, June 1969.

6. Dickson, L. E., *History of the Theory of Numbers*, Chelsea Publishing Co., New York, 1952.

7. Fadeeva, V. N., *Computational Methods of Linear Algebra*, Dover Publications, Inc., New York, 1959.

8. Gauss, C. F., *Disquisitiones Arithmeticae*, translated by A. A. Clarke, Yale University Press, New Haven, 1966.

9. Gauss, C. F., Nachlass Werke, III, "Theoria Interpolationis Methodo Nova Tractata" (Göttingen, 1876), p. 265.

10. Greenstreet, W. J. (Editor), *Isaac Newton 1642-1727*, G. Bell and Sons, Ltd., London, 1927.

11. Knuth, D. E., *The Art of Computer Programming*, Vol. 2 (Seminumerical Algorithms), Addison-Wesley Publishing Company, Inc., Reading, Mass., 1969.

12. Lagrange, J.-L., *Oeuvres*, Vol. 7 (Les Lecons Elementaires sur les Mathematiques - Lecon Cinquieme: Sur l'usage des Courbes dans la solution des Problèmes, p. 286 (1795)).

13. MacLane, S., and G. Birkhoff, *Algebra*, The Macmillan Company, New York, 1967.

14. Newman, M., "Solving Equations Exactly," *J. Res. Nat. Bur. Standards*, Vol. 71B, No. 4 (Oct.-Dec. 1967), pp. 171-179.

15. Newton, I., *Mathematical Principles of Natural Philosophy and his System of the World*, Revised Translation by F. Cajori, University of California Press, Berkeley, California, 1960.

16. Ore, O., *Number Theory and its History*, McGraw-Hill Book Co., New York, 1948.

17. Szabo, N. S., and R. I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology*, McGraw-Hill Book Company, New York, 1967.

18. Takahasi, H. and Y. Ishibashi, "A New Method for Exact Calculation by a Digital Computer," *Information Processing in Japan*, Vol. 1 (1961), pp. 28-42.

19. Waring, E., "Problems Concerning Interpolations," *Phil. Trans. of the Royal Soc. of London*, Vol. 59, (1779), pp. 59-67.

X

B